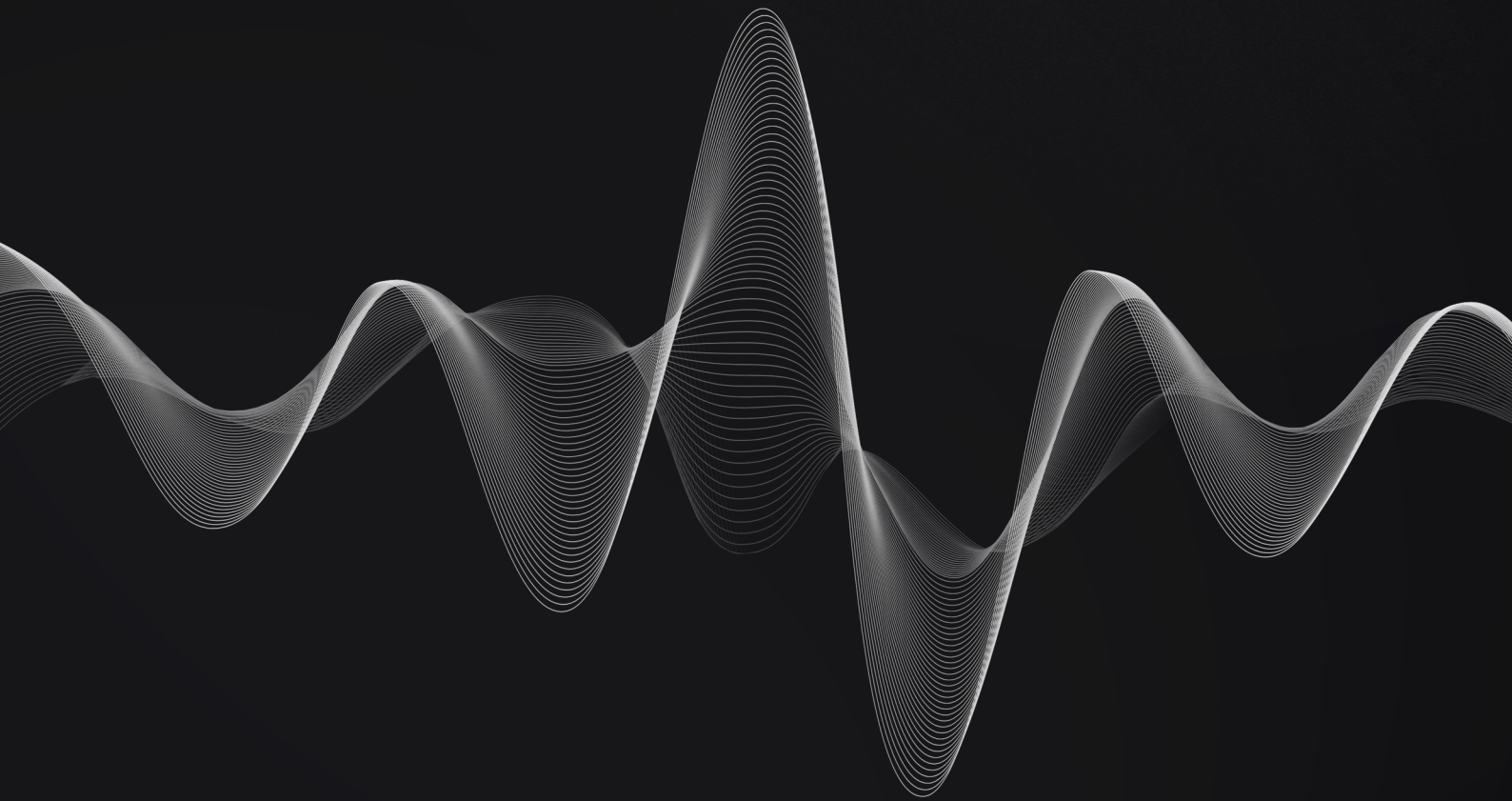




# DeFi-Dynamics

## Prototype Audit Report









# Document Control

**PUBLIC**

**FINAL**(v1.6)

## Audit\_Report\_DeFi-DY-SC01\_FINAL\_16

May 30, 2023		v0.1	Engineer 1: Initial Draft
Jun 8, 2023		v0.3	Engineer 2: Added Findings
Jun 8, 2023		v0.4	Engineer 3: Added Findings
Jun 12, 2023		v0.5	Engineer 1, Engineer 2, Engineer 3: Final Draft
Jun 13, 2023		v0.6	PM: Approved
Jun 13, 2023		v1.0	Executive Management: Approved

<b>Points of Contact</b>	John Defiway CEO PM	DeFi-Dynamics Resonance Resonance	j.defiway@defi-dynamics.io ceo@resonance.security pm@resonance.security
--------------------------	---------------------------	---	---

<b>Testing Team</b>	Engineer 1 Engineer 2 Engineer 3	Resonance Resonance Resonance	engineer1@resonance.security engineer2@resonance.security engineer3@resonance.security
---------------------	--	-------------------------------------	--

## Copyright and Disclaimer

© 2023 Resonance Security, LLC. All rights reserved.

**This report is an illustrative prototype of an audit report example provided by Resonance Security, LLC. It does not reflect a comprehensive security analysis of a specific target system or codebase. Please be advised this is not an exhaustive list of vulnerabilities and should be considered an exclusive sample and property of Resonance Security LLC.**

The information in this report is considered confidential and proprietary by Resonance and is licensed to the recipient solely under the terms of the project statement of work. Reproduction or distribution, in whole or in part, is strictly prohibited without the express written permission of Resonance.

All activities performed by Resonance in connection with this project were carried out in accordance with the project statement of work and agreed-upon project plan. It's important to note that security assessments are time-limited and may depend on information provided by the client, its affiliates, or partners. As such, the findings documented in this report should not be considered a comprehensive list of all security issues, flaws, or defects in the target system or codebase.

Furthermore, it is hereby assumed that all of the risks in electing not to remedy the security issues identified henceforth are sole responsibility of the respective client. The acknowledgement and understanding of the risks which may arise due to failure to remedy the described security issues, waives and releases any claims against Resonance, now known or hereafter known, on account of damage or financial loss.

# Contents

<b>1 Document Control</b>	<b>2</b>
Copyright and Disclaimer .....	2
<b>2 Executive Summary</b>	<b>4</b>
System Overview .....	4
Repository Coverage and Quality.....	4
<b>3 Target</b>	<b>6</b>
<b>4 Methodology</b>	<b>7</b>
Severity Rating.....	8
Repository Coverage and Quality Rating.....	9
<b>5 Findings</b>	<b>10</b>
Lack of Access Control in a Key Contract Functions.....	11
Reentrancy through deposit() Function.....	12
Unreachable Code After Return Statement in withdrawFunds() Function .....	14
A malicious user can create a strategy that is actually empty.....	15
Casting Overflow.....	16
Permanent Lock of Funds in Absence of Bet Cancellation Functionality.....	17
Sandwich Attacks Due to Lack of Slippage Checks .....	18
Lake of Pausing Mechanisms Utilizations.....	19
Missing Two-Step Ownership Transfer .....	20
Payble function allows for Eth transfer even when ERC20 tokens are being used .....	21
Inconsistent Function Behavior in ERC1155 Token Handling .....	22
Unnecessary Initialization Of Variables With Default Values .....	23
No Usage Of OpenZeppelin's Math Library .....	24
<b>A Proof of Concepts</b>	<b>25</b>

# Executive Summary

**DeFi-Dynamics** contracted the services of Resonance to conduct a comprehensive security audit of their smart contracts **between June 13, 2023 and June 23, 2023**. The primary objective of the assessment was to identify any potential security vulnerabilities and ensure the correct functioning of smart contract operations.

During the engagement, Resonance allocated **3** engineers to perform the security review. The engineers, including an accomplished professional with extensive proficiency in blockchain and smart-contract security, encompassing specialized skills in advanced penetration testing, and in-depth knowledge of multiple blockchain protocols, devoted **7 days** to the project. The project's test targets, overview, and coverage details are available throughout the next sections of the report.

The ultimate goal of the audit was to provide **DeFi-Dynamics** with a detailed summary of the findings, including any identified vulnerabilities, and recommendations to mitigate any discovered risks. The results of the audit are presented in detail further below.



## System Overview

DeFi-Dynamics is a comprehensive **DeFi platform** offering an extensive range of financial products and services. Built on Ethereum, the system leverages smart contracts to enable various decentralized finance operations such as yield farming, liquidity mining, staking, and swapping.

**Another service DeFi-Dynamics offers is its NFT Marketplace.** This is a decentralized marketplace where users can mint, buy, sell, and trade non-fungible tokens (NFTs). It provides an inclusive space for artists, creators, and collectors to interact directly with one another, eliminating the need for intermediaries. Users can transact with a wide variety of NFTs, including art, virtual real estate, collectibles, and more, all empowered by Ethereum's ERC-721 and ERC-1155 token standards.

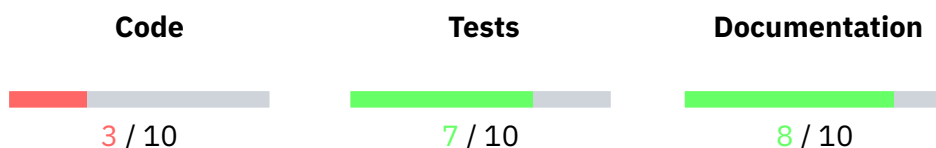
The Automated Portfolio Optimizer (APO) is another key component, assisting users to maximize returns through strategic asset reallocation across diverse yield farming strategies and liquidity pools. It operates based on real-time market data and advanced predictive algorithms.

The platform's governance is driven by the **DeFiDynamicsToken (DFD), an ERC-20 token**. DFD token holders can propose and vote on changes to the system parameters, facilitating a truly community-led project.

Finally, DeFi-Dynamics is designed to integrate seamlessly with a host of other DeFi protocols and stablecoins. This interoperability fosters a robust ecosystem capable of supporting complex financial transactions, thereby advancing the financial inclusivity mission of DeFi.



## Repository Coverage and Quality



Resonance's testing team has assessed the Code, Tests, and Documentation coverage and quality of the DeFi-Dynamics Web3 System and achieved the following results:

- The code: The code structure of the DeFi-Dynamics Web3 system, though functional, does not adhere to the recommended development best practices nor utilizes standard libraries and language guides. The readability suffers as a result, making it challenging for external contributors or auditors to understand. It is also observed that the latest stable versions of the associated components are not in use. As a result, the overall code quality is assessed as substandard and warrants significant improvements.
- Unit and Integration Tests: Despite the shortcomings in the code structure, the inclusion of unit and integration tests is commendable. The tests adequately cover both the technical and functional requirements of the system. With a code coverage of 64%, the overall testing quality is deemed good. However, efforts should be made to increase the code coverage to ensure the resilience of the system under various conditions.
- The documentation: The documentation provided does a satisfactory job in specifying the system's overall workings, detailing the technical aspects of the code, and explaining the workflows and interactions. However, the lack of detailed explanations and step-by-step guides brings down its effectiveness. As a result, while the documentation quality is passable, there is ample room for improvement to enhance the system's understandability and maintainability.

# Target

The objective of this project is to conduct a comprehensive review and security analysis of the smart contracts that are contained within the specified repository.

The following items are included as targets of the security assessment:

- Repository: DeFi-Dynamics/Web3System-contracts-v1
- Hash: 9f7sd8f6s9df7s6d8f7s6df87sd6f8s76d

The following items are excluded:

- External and standard libraries
- Files pertaining to the deployment process

# Methodology

In the context of security audits, Resonance's primary objective is to portray the workflow of a real-world cyber attack against an entity or organization, and document in a report the findings, vulnerabilities, and techniques used by malicious actors. While several approaches can be taken into consideration during the assessment, Resonance's core value comes from the ability to correlate automated and manual analysis of system components and reach a comprehensive understanding and awareness with the customer on security-related issues.

Resonance implements several and extensive verifications based off industry's standards, such as, identification and exploitation of security vulnerabilities both public and proprietary, static and dynamic testing of relevant workflows, adherence and knowledge of security best practices, assurance of system specifications and requirements, and more. Resonance's approach is therefore consistent, credible and essential, for customers to maintain a low degree of risk exposure.

Ultimately, product owners are able to analyze the audit from the perspective of a malicious actor and distinguish where, how, and why security gaps exist in their assets, and mitigate them in a timely fashion.

## Source Code Review - Solidity EVM

During source code reviews for Web3 assets, Resonance includes a specific methodology that better attempts to effectively test the system in check:

1. Review specifications, documentation, and functionalities
2. Assert functionalities work as intended and specified
3. Deploy system in test environment and execute deployment processes and tests
4. Perform automated code review with public and proprietary tools
5. Perform manual code review with several experienced engineers
6. Attempt to discover and exploit security-related findings
7. Examine code quality and adherence to development and security best practices
8. Specify concise recommendations and action items
9. Revise mitigating efforts and validate the security of the system

Additionally and specifically for Solidity EVM audits, the following attack scenarios and tests are recreated by Resonance to guarantee the most thorough coverage of the codebase:

- Reentrancy attacks
- Frontrunning attacks
- Unsafe external calls
- Unsafe third party integrations
- Denial of service
- Access control issues



- Inaccurate business logic implementations
- Incorrect gas usage
- Arithmetic issues
- Unsafe callbacks
- Timestamp dependence
- Mishandled panics, errors and exceptions



## Severity Rating

Security findings identified by Resonance are rated based on a Severity Rating which is, in turn, calculated off the **impact** and **likelihood** of a related security incident taking place. This rating provides a way to capture the principal characteristics of a finding in these two categories and produce a score reflecting its severity. The score can then be translated into a qualitative representation to help customers properly assess and prioritize their vulnerability management processes.

The **impact** of a finding can be categorized in the following levels:

1. Weak - Inconsequential or minimal damage or loss
2. Medium - Temporary or partial damage or loss
3. Strong - Significant or unrecoverable damage or loss

The **likelihood** of a finding can be categorized in the following levels:

1. Unlikely - Requires substantial knowledge or effort or uncontrollable conditions
2. Likely - Requires technical knowledge or no special conditions
3. Very Likely - Requires trivial knowledge or effort or no conditions

		Likelihood		
		Very Likely	Likely	Unlikely
Impact	Strong	Critical	High	Medium
	Medium	High	Medium	Low
	Weak	Medium	Low	Info





# Repository Coverage and Quality Rating

The assessment of Code, Tests, and Documentation coverage and quality is one of many goals of Resonance to maintain a high-level of accountability and excellence in building the Web3 industry. In Resonance it is believed to be paramount that builders start off with a good supporting base, not only development-wise, but also with the different security aspects in mind. A product, well thought out and built right from the start, is inherently a more secure product, and has the potential to be a game-changer for Web3's new generation of blockchains, smart contracts, and dApps.

Accordingly, Resonance implements the evaluation of the code, the tests, and the documentation on a score **from 1 to 10** (1 being the lowest and 10 being the highest) to assess their quality and coverage. In more detail:

- Code should follow development best practices, including usage of known patterns, standard libraries, and language guides. It should be easily readable throughout its structure, completed with relevant comments, and make use of the latest stable version components, which most of the times are naturally more secure.
- Tests should always be included to assess both technical and functional requirements of the system. Unit testing alone does not provide sufficient knowledge about the correct functioning of the code. Integration tests are often where most security issues are found, and should always be included. Furthermore, the tests should cover the entirety of the codebase, making sure no line of code is left unchecked.
- Documentation should provide sufficient knowledge for the users of the system. It is useful for developers and power-users to understand the technical and specification details behind each section of the code, as well as, regular users who need to discern the different functional workflows to interact with the system.

# Findings

During the security audit, several findings were identified to possess a certain degree of security-related weaknesses. These findings, represented by unique IDs, are detailed in this section with relevant information including Severity, Category, Status, Code Section, Description, and Recommendation. Further extensive information may be included in corresponding appendices should it be required.

An overview of all the identified findings is outlined in the table below, where they are sorted by Severity and include a **Remediation Priority** metric asserted by Resonance's Testing Team. This metric characterizes findings as follows:

- "Quick Win" Requires little work for a high impact on risk reduction.
- "Standard Fix" Requires an average amount of work to fully reduce the risk.
- "Heavy Project" Requires extensive work for a low impact on risk reduction.

---

Findings ID	Description	Severity	Status
RES-01	Lack of Access Control in a Key Contract Functions	●●●●	Unresolved
RES-02	Reentrancy through deposit() Function	●●●●	Resolved
RES-03	Unreachable Code After Return Statement in withdraw-Funds() Function	●●●●	Unresolved
RES-04	A malicious user can create a strategy that is actually empty	●●●●	Unresolved
RES-05	Casting Overflow	●●●●	Resolved
RES-06	Permanent Lock of Funds in Absence of Bet Cancellation Functionality	●●●●	Acknowledged
RES-07	Sandwich Attacks Due to Lack of Slippage Checks	●●●●	Unresolved
RES-08	Lake of Pausing Mechanisms Utilizations	●●●●	Acknowledged
RES-09	Missing Two-Step Ownership Transfer	●●●●	Resolved
RES-10	Payble function allows for Eth transfer even when ERC20 tokens are being used	●●●●	Acknowledged
RES-11	Inconsistent Function Behavior in ERC1155 Token Handling	●●●●	Resolved
RES-12	Unnecessary Initialization Of Variables With Default Values	●●●●	Unresolved
RES-13	No Usage Of OpenZeppelin's Math Library	●●●●	Unresolved



# Lack of Access Control in a Key Contract Functions

Critical

RES-DeFi-DY-SC01-01

Access Control

Unresolved

## Code Section

- [contracts/GenericLendingPool.sol#L11](#)
- [contracts/GenericLendingPool.sol#L54](#)
- [contracts/GenericLendingPool.sol#L98](#)
- [contracts/GenericLendingPool.sol#L233](#)

## Description

Our audit has identified significant vulnerabilities within the lending protocol smart contract, specifically the `GenericLendingPool` contract. Management functions of the contract are publicly exposed without any access control. They include:

- `addPool1`: This function can add any lending pool into the protocol without restrictions.
- `addAsset`: This function allows any asset, including those not whitelisted or vetted, to be added into any lending pool.
- `updateWeight`: This function permits updating the weight of any asset within a lending pool, potentially disrupting the pool's balance.
- `setWeight`: Similar to `updateWeight`, but this function applies to all assets within a lending pool.

## Recommendation

We strongly recommend the introduction of adequate access controls for these functions to prevent potential exploitation. Given the nature of DeFi and the potential for considerable financial loss, unrestricted access to these functions could allow malicious actors to manipulate the protocol in their favor.

Implementing an access control mechanism, such as a multi-signature requirement for calling these sensitive functions, could drastically enhance the security of the protocol. This will prevent single addresses from exerting undue influence over the protocol and ensure a more decentralized control over key parameters.

Additionally, whitelisting vetted assets and pools could prevent the addition of potentially risky or fraudulent entities, thereby further improving the overall security posture of your lending protocol.



# Reentrancy through deposit() Function

Critical

RES-DeFi-DY-SC01-02

Business Logic

Resolved

## Code Section

- [contracts/vaults/InvestmentVault.sol#L150-L160](#)

## Description

A Reentrancy attack occurs when an external contract hijacks the control flow, and re-enters the targeted contract **before the previous interactions have finished**. Here, the `deposit()` function in the `InvestmentVault.sol` contract of DeFi-Dynamics platform can lead to a potential reentrancy attack if the token is an **ERC777** token, which allows the sender to retain control.

In the `InvestmentVault.sol` contract, during the `deposit()` function, the contract's balance is cached before a `token.transferFrom()` is called. This could lead to exploits if the token is a type that gives control to the sender, like ERC777 tokens.

```
function deposit(uint256 _amount) public noContract(msg.sender) { //  
    ↪ InvestmentVault.sol:L150-L160  
  
    require(_amount > 0, "INVALID_AMOUNT");  
  
    uint256 balanceBefore = balance();  
  
    token.safeTransferFrom(msg.sender, address(this), _amount);  
  
    uint256 supply = totalSupply();  
  
    uint256 shares;  
  
    if (supply == 0) {  
  
        shares = _amount;  
  
    } else {  
  
        shares = (_amount * supply) / balanceBefore;  
  
    }  
  
    _mint(msg.sender, shares);  
  
    emit Deposit(msg.sender, _amount);  
  
}
```

## Recommendation

To mitigate this vulnerability, the `token.safeTransferFrom()` should be moved as the last call in `deposit()` function. By doing so, the entire function completes before any external contracts (like ERC777) can interact with it, thus avoiding the risk of a reentrancy attack.

```
function deposit(uint256 _amount) public noContract(msg.sender) {  
  
    require(_amount > 0, "INVALID_AMOUNT");  
  
    uint256 supply = totalSupply();  
  
    uint256 shares;  
  
    if (supply == 0) {  
  
        shares = _amount;  
  
    } else {  
  
        uint256 balanceBefore = balance();  
  
        shares = (_amount * supply) / balanceBefore;  
  
    }  
  
    _mint(msg.sender, shares);  
  
    token.safeTransferFrom(msg.sender, address(this), _amount); // Moved this line to  
    ↪ the last  
  
    emit Deposit(msg.sender, _amount);  
  
}
```

## Status

*The issue has been fixed in fde6az83254d2c35e579d0bee063af0a7xpdf4e.*



# Unreachable Code After Return Statement in withdrawFunds() Function

High

RES-DeFi-DY-SC01-03

Access Control

Unresolved

## Code Section

- [contracts/strategies/PortfolioStrategies.sol#L214-L216](#)

## Description

The `PortfolioStrategies.sol` contract in the DeFi-Dynamics platform exhibits an issue due to certain development practices. Specifically, the `withdrawFunds()` function contains a return statement followed by a `require` check.

```
(bool success, ) = [msg.sender.call](http://msg.sender.call){value: _amount}(""); //  
↪ PortfolioStrategies.sol:L214  
  
return _amount;  
  
require(success, "Transfer failed");
```

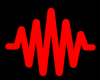
The `require` check, designed to ensure that the transfer was successful, will never be reached due to the `return` statement ending the execution prematurely. Consequently, the token transfer will always be considered successful, irrespective of the actual transaction result.

## Recommendation

To address this vulnerability, the lines 214 and 215 in `PortfolioStrategies.sol` should be inverted as follows:

```
(bool success, ) = [msg.sender.call](http://msg.sender.call){value: _amount}("");  
  
require(success, "Transfer failed");  
  
return _amount;
```

This adjustment ensures that the `require` check is executed prior to the `return` statement, validating the transfer before the function execution concludes. This change would help maintain the contract's integrity by not falsely portraying unsuccessful transfers as successful.



# A malicious user can create a strategy that is actually empty

High

RES-DeFi-DY-SC01-04

Data Validation

Unresolved

## Code Section

- [contracts/strategies/PortfolioStrategies.sol#L197-L200](#)

## Description

In the `PortfolioStrategies.sol` contract, a malicious user can call the `createStrategy()` function to create a strategy with an ERC20 token that returns false rather than revert on failed transfer. By specifying the `tokenType` parameter to be `ERC721`, the `transferFrom()` function will be called rather than `safeTransferFrom()`. This results in potentially creating a strategy with a high `tokenValue` but empty of actual assets.

This could trick other users into buying this strategy if they do not confirm that the token and `tokenType` on the strategy actually match.

File: `PortfolioStrategies.sol`

```
197: // transfer the NFTs or ERC20s to the contract

198: strategy.tokenType == TokenType.ERC721

199: ? ERC721(strategy.token).transferFrom(msg.sender, address(this),
    ↪ strategy.tokenValue)

200: : ERC20(strategy.token).safeTransferFrom(msg.sender, address(this),
    ↪ strategy.tokenValue);
```

## Recommendation

The most straightforward solution is to use `safeTransferFrom()` when the token is of type `ERC721`. Since the transfer occurs at the end of the function, there shouldn't be any risk of reentrancy. If someone passes an ERC20 address with type `ERC721`, the `safeTransferFrom()` call would simply fail since that function signature shouldn't exist on ERC20 tokens.





# Casting Overflow

High

RES-DeFi-DY-SC01-05

Arithmetic Issues

Resolved

## Code Section

- [contracts/funding/contract1.sol#L33](#)
- [contracts/funding/contract2.sol#L332-L333](#)

## Description

In situations where a variable with a larger bit representation (e.g., uint256) is cast down to a variable with a smaller bit representation (e.g., uint128), the possibility of casting overflow exists. This type of overflow isn't automatically detected or reverted by the Solidity language or the Ethereum Virtual Machine (EVM), thus manual accounting is required in the smart contract's source code.

In the audited smart contract, the functions `function1()` and `functions2()` were identified as being susceptible to such casting overflows.

## Recommendation

To mitigate the risk of casting overflow, it is highly recommended that arithmetic operations are performed on variables that have the same bit representation. Additionally, it is advisable to include manual checks in the source code to account for instances when variable cast downs are unavoidable, thus preventing possible casting overflows.

## Status

*The issue has been fixed in `fde6az83254d2c35e579d0bee063af0a7x4df4e`.*



# Permanent Lock of Funds in Absence of Bet Cancellation Functionality

High

RES-DeFi-DY-SC01-06

Business Logic

Acknowledged

## Code Section

- [contracts/market/DeFi-DynamicsNFTmarket.sol#L55](#)

## Description

A significant security vulnerability has been found in the DeFi-Dynamics NFT marketplace. The `makeOffer()` function, which allows users to bet on digital assets (akin to rare office items), can result in permanent fund lock due to the absence of a bet cancellation function.

To place a bet on a digital asset (e.g., a "Golden Trader Portfolio"), a user must call the `makeOffer()` function with the desired amount of ether. The DeFi-Dynamics contract then stores this ether in escrow until the asset owner accepts the offer. As a result, the DeFi-Dynamics contract must hold the total value of any unresolved offers in its balance.

The critical issue is that there's no functionality for a bidder to cancel their offer, which can result in a permanent lock of funds if the asset owner never chooses to accept the offer.

### Threat Scenario:

Consider a situation where a user from DeFi-Dynamics wishes to bid on a rare "Golden Trader Portfolio" item and places an offer by calling the `makeOffer()` function. Suppose the owner of the "Golden Trader Portfolio" never accepts the offer. In this case, the funds that the bidder used for the offer will remain locked indefinitely, as there's no mechanism to cancel the offer and retrieve the funds.

## Recommendation

**Implement a function to allow users to cancel their offers**, effectively retrieving their locked funds. This feature will add an extra layer of security and flexibility for users, as it ensures they have control over their assets even after an offer has been made. The absence of such a function presents a risk of funds getting permanently locked, which could lead to user dissatisfaction and potential abandonment of the platform. Thus, introducing a bid cancellation function is critical to maintaining user trust and platform integrity.



# Sandwich Attacks Due to Lack of Slippage Checks

High

RES-DeFi-DY-SC01-07

Data Validation

Unresolved

## Code Section

- [contracts/strategies/PortfolioStrategies.sol#L197-L200](#)

## Description

Sandwich attacks are a form of frontrunning attack that involves manipulating the price of an asset for profit. The attacker observes a large transaction in the mempool, places a transaction before it to drive the price up, and another one after it to sell at the inflated price.

This type of attack can occur in DeFi protocols that don't check for slippage, allowing the price to change significantly between when a transaction is signed and when it's mined.

DeFi-Dynamics's protocol is susceptible to sandwich attacks due to lack of slippage checks in the `swapTokensForTokens()` function. Here is a potential threat scenario:

1. Alice initiates a large transaction intending to swap a significant amount of Token A for Token B using the `swapTokensForTokens()` function.
2. Bob, an attacker, observing the mempool, identifies Alice's large transaction waiting to be mined.
3. Bob executes a buy order for Token B before Alice's transaction, significantly inflating the price due to the large order size. He does this by submitting a transaction with a higher gas price.
4. Alice's transaction gets mined next, buying Token B at the inflated price set by Bob's transaction.
5. Bob then executes a sell order for Token B after Alice's transaction, benefiting from the inflated price.

The result is that Alice pays significantly more for Token B than expected, while Bob profits from the artificially inflated price.

## Recommendation

To prevent such sandwich attacks, DeFi-Dynamics could implement slippage checks in their smart contracts to ensure the final price of a transaction does not deviate too much from the expected price. If the price deviation exceeds a pre-set tolerance, the transaction could be reverted to protect users from price manipulation. In addition, DeFi-Dynamics could consider implementing a time-lock or delay on large transactions to give users the opportunity to cancel their transactions if abnormal price movement is detected.



# Lake of Pausing Mechanisms Utilizations

Medium

RES-DeFi-DY-SC01-08

Access Control

Acknowledged

## Code Section

- [contracts/market/DeFi-DynamicsFunding.sol#L148](#)
- [contracts/market/DeFi-DynamicsFunding.sol#L191](#)
- [contracts/market/DeFi-DynamicsFunding.sol#L311](#)
- [contracts/tokens/DeFi-DynamicsTokens.sol#L292](#)

## Description

The DeFi-Dynamics platform's smart contracts primarily inherit from the `ProtocolGovernanceAccess` contract. This contract in turn uses OpenZeppelin's `Pausable` functionality, designed to halt critical operations during emergencies.

Nonetheless, critical operations such as `addLiquidity()` and `createStrategy()`, found in the `LiquidityPool` and `StrategyCreator` contracts respectively, lack implementation of the `Pausable` functionality. For instance, a flaw in the `addLiquidity()` function that allows limitless token input could inflate the liquidity pool size, destabilizing token prices.

If pausing was correctly implemented, DeFi-Dynamics **could halt** `addLiquidity()` upon discovering the flaw, preventing new liquidity until the flaw is remedied. Without `Pausable` in place, the platform remains vulnerable until the flaw is fixed and deployed.

## Recommendation

It is highly recommended to implement code logic that prevents these crucial actions from occurring during system emergencies. For this purpose, the smart contract's pausing modifiers `whenPaused` and `whenActive` should be used to prevent the platform state from transitioning into an unstable state during emergencies.

Implementing such measures will allow for greater control during critical moments, preventing transactions that could lead to financial losses or disrupt the functioning of the system.



# Missing Two-Step Ownership Transfer

Medium RES-DeFi-DY-SC01-09

Business Logic

Resolved

## Code Section

- [contracts/platform/DeFiDynamicsPlatform.sol#L230-245](#)

## Description

During our audit of the `DeFiDynamicsPlatform.sol` contract, we identified a potential vulnerability in the `transferGovernanceRole()` function. This function is responsible for transferring the critical role of `GovernanceRole`, which is initially assigned to the contract's deployer and has control over platform management including access control, function permissions, and more.

The function `transferGovernanceRole()` executes the transfer of the `GovernanceRole` role in a single step. This approach can be risky for such a crucial operation because an error in transferring this role to an erroneous address could make the entire system inoperable.

### Threat Scenario:

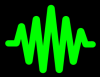
Consider if Jordan Smith, the original CTO and deployer of DeFi-Dynamics contracts, decides to transfer his role to a new manager using `_transferGovernanceRole()`. If Jordan **inadvertently** inputs an incorrect address, the system could become entirely inaccessible, rendering the entire DeFi-Dynamics platform **inoperative**.

## Recommendation

To mitigate this vulnerability, we recommend implementing a two-step process for transferring the essential `GovernanceRole` role. The first step could be the nomination of a new manager by the current `GovernanceRole` holder. The second step should be the acceptance of this nomination by the nominated address. This two-step process provides an additional layer of security and ensures that the system remains functional, even if an error occurs during the transfer. This can be achieved by adding a new function called `nominateNewGovernanceRole(address)` and modifying the current `_transferGovernanceRole()` to only allow the nominated address to accept the role transfer.

## Status

*The issue has been fixed in `fde6az83254d2c35e579d0bee063af0a7xpdf4e`.*



# Payble function allows for Eth transfer even when ERC20 tokens are being used

Low

RES-DeFi-DY-SC01-10

Data Validation

Acknowledged

## Code Section

- [contracts/vaults/PortfolioVault.sol#L78-L80](#)

## Description

In DeFi-Dynamics platform's `PortfolioVault.sol` contract, the function `depositAsset()` is payable. It needs to be payable for cases where the function `_depositToYieldPool()` expects ETH to be transferred.

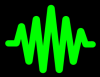
However, when `PortfolioVault.sol` is not using Ether as an asset, and `msg.value` is greater than 0, any Ether value attached to the transaction will be stuck in the contract.

## Recommendation

To mitigate this vulnerability, checks should be added to `_depositToYieldPool()` function in the `PortfolioStrategies.sol` to require `msg.value == 0` when the asset being deposited isn't Ether. Similarly, checks should be added in the case where asset is not Ether to require `msg.value == 0`.

Changes in the contract `PortfolioStrategies.sol`:

These changes would prevent inadvertent Ether transfers in transactions primarily dealing with ERC20 tokens, thus avoiding the issue of trapped Ether.



# Inconsistent Function Behavior in ERC1155 Token Handling

Low

RES-DeFi-DY-SC01-11

Data Validation

Resolved

## Code Section

- [contracts/tokens/DeFiDynamicsTokens.sol#L292-319](#)

## Description

The functions `onERC1155Received()` and `onERC1155BatchReceived()` in the `DeFiDynamicsTokens` contract perform similar functionalities, however, their actual behaviors differ subtly, which could lead to confusion and potential issues. Both functions are intended to handle incoming ERC1155 tokens, but they handle the tokens differently.

The `onERC1155BatchReceived()` function includes an **additional validation** of the sender (`from`) argument, making it successful only for token minting operations. On the other hand, the `onERC1155Received()` function **does not have this additional validation**, allowing it to successfully execute both for minting and transferring operations.

This discrepancy creates ambiguity for users and contract interactors, who may choose to use one function over the other without fully understanding the differences in their behavior.

## Recommendation

To avoid potential issues and confusion, it is recommended that functions with similar intended behavior should perform the same set of actions, be they function calls, data validations, event emissions, or error handling (reverts). This would help maintain consistency and transparency in contract operations, enhancing both security and usability.

## Status

*The issue has been fixed in `fde6az83254d2c35e579d0bee063af0a7x4df4e`.*





# Unnecessary Initialization Of Variables With Default Values

Info

RES-DeFi-DY-SC01-12

Gas Optimization

Unresolved

## Code Section

- Not specified

## Description

In the Solidity programming language, **all variables are automatically initialized to a default value corresponding to their type when they are declared**. For example, integer types are initialized to 0, boolean types to `false`, and address types to `0x00`. Explicitly initializing variables to these default values when they are declared is therefore redundant, and since each operation in a contract costs gas, it results in unnecessary gas costs. This could potentially impact the contract's efficiency and the cost of executing its functions. It's important to review the contract's code to identify any instances of this issue and optimize for gas efficiency.

## Recommendation

Review your contract's code for variable declarations where the variable is explicitly initialized to the type's default value. Remove the explicit initialization and let Solidity automatically initialize the variable. Here's an example:

Before:

```
uint256 public counter = 0;
```

After:

```
uint256 public counter;
```

In both cases, `counter` will be initialized to 0. However, the latter declaration will cost less gas when the contract is deployed, improving the contract's efficiency.



# No Usage Of OpenZeppelin's Math Library

Info

RES-DeFi-DY-SC01-13

Code Quality

Unresolved

## Code Section

- [contracts/Math.sol](#)

## Description

OpenZeppelin implements several standards used all across blockchain development, especially in Solidity and the Ethereum Virtual Machine. These standards primary goal is to unify and normalize development patterns so that the entire blockchain may be more understandable and readily usable.

The Ubet Betting Platform implements source code that mimics or is very similar to components already implemented by OpenZepellin, and could therefore, be switched with the more the standard well-known approach. Such source code includes:

- Function `ceildiv()` on `Math` library, can be substituted with OpenZeppelin's implementation on `Math` library.
- Function `min()` on `Math` library, can be substituted with OpenZeppelin's implementation on `Math` library.

It should be noted that the usage of bigger libraries does not constitute an overuse of gas, since only the functions that are used are included on the bytecode of the smart contract.

## Recommendation

It is recommended to make use of implemented and audited standards that already solve the necessary functionalities.

# Proof of Concepts

## RES-02 Reentrancy through deposit() Function

```
// Initialize foundry, provider, and signer

const foundry = new Foundry();

const provider = new ethers.providers.JsonRpcProvider();

const signer = provider.getSigner();

// Initialize DeFi-Dynamics contract

const contractAddress = "<contract_address>";

const abi = <contract_abi>;

const contract = new ethers.Contract(contractAddress, abi, signer);

// Outer deposit of 500 tokens

await contract.deposit(500, {from: attackerAddress});

// Inner deposit of 500 tokens triggered by the reentrancy exploit

// This would be implemented within the malicious contract's code

// For this PoC, we will simulate it by making a second call to deposit

await contract.deposit(500, {from: attackerAddress});

// At this point, the attacker has deposited a total of 1000 tokens

// But due to the reentrancy exploit, they have received 1250 shares in return

// The attacker can now withdraw these shares to make a profit

await contract.withdraw(1250, {from: attackerAddress});

// Calculate profit

const finalBalance = await contract.balance();

const profit = finalBalance - 1000;

console.log(`Profit made: ${profit} tokens`);
```

## RES-04 A malicious user can create a strategy that is actually empty

```
import { ethers, Foundry } from "@openzeppelin/hardhat-foundry";

import { ERC721, PortfolioStrategies } from
↳ "@defi-dynamics/defi-dynamics-contracts";

// Initialize environment

await Foundry.snapshot();

// Initialize contracts

const strategiesContract = await Foundry.getDeployedContract("PortfolioStrategies");

const erc721Contract = await Foundry.getDeployedContract("ERC721");

const erc20Contract = await Foundry.getDeployedContract("ERC20");

// Initialize signer

const [signer] = await ethers.getSigners();

// Mint ERC721 and ERC20 tokens

await [erc721Contract.mint](http://erc721Contract.mint)(signer.address, 1);

await [erc20Contract.mint](http://erc20Contract.mint)(signer.address, 1000);

// Approve PortfolioStrategies to manage ERC721 and ERC20 tokens

await erc721Contract.setApprovalForAll(strategiesContract.address, true);

await erc20Contract.approve(strategiesContract.address, 1000);

// Call createStrategy with TokenType set to ERC721 but pass ERC20 token address

await strategiesContract.connect(signer).createStrategy(

erc20Contract.address, // Incorrect token address

1, // TokenType.ERC721

1, // tokenValue

ethers.utils.parseEther("1"), // strategy price

ethers.constants.MaxUint256, // deadline

"False ERC721 strategy" // name

);
```

```

// Retrieve created strategy details

const strategy = await strategiesContract.getStrategy(1);

console.log(`Strategy ID: ${[strategy.id] (http://strategy.id)}`);

console.log(`Token address: ${[strategy.token]}`);

console.log(`Token type: ${[strategy.tokenType]}`);

console.log(`Token value: ${[strategy.tokenValue.toString()]}`);

console.log(`Strategy price: ${[ethers.utils.formatEther(strategy.price)]}`);

console.log(`Strategy name: ${[strategy.name] (http://strategy.name)}`);

```

## RES-09 Missing Two-Step Ownership Transfer

```

import { DeFiDynamicsPlatform, ethers } from "@openzeppelin/defi-dynamics-platform";

// Initialize web3 provider and contract instances

const provider = new ethers.providers.JsonRpcProvider("<http://localhost:8545>");

const signer = provider.getSigner();

const defiDynamicsPlatform = new DeFiDynamicsPlatform(signer);

// Get the initial _GovernanceRole address (should be the contract deployer's
→ address)

const initialGovernanceRole = await defiDynamicsPlatform.governanceRole();

console.log(`Initial Governance Role: ${[initialGovernanceRole]}`);

// Mistakenly transfer _GovernanceRole role to incorrect address

const incorrectAddress = "0x0000000000000000000000000000000000000000dead";

const tx = await defiDynamicsPlatform.transferGovernanceRole(incorrectAddress);

await tx.wait();

// Try to verify the _GovernanceRole change

const newGovernanceRole = await defiDynamicsPlatform.governanceRole();

console.log(`New Governance Role: ${[newGovernanceRole]}`);

if (newGovernanceRole === incorrectAddress) {

```

```
console.error(`Critical Error: Governance Role was incorrectly transferred to  
↳  ${incorrectAddress}`);  
  
console.error("The platform's functions are now inaccessible, rendering the system  
↳  inoperable.");  
  
}
```